C Language: Review Notes

Feiyi Wang

October, 2008

Contents

1 Language Basics

Why C: (1) C provides a fairly complete set of facilities (data type, structures, pointers, string, control structure, run-time library) to deal with various applications. (2) C programs are efficient: a small semantic gap between C and computer hardware (3) C programs are quite portable across different system.

1.1 Data Types

Valid data types are:

- `int`, if the first digit is 0, then it is taken as octal notation, base is 8. and hex expression would be 0x of course.
- `float`, To display floating point value at the terminal, the `printf` conversion character `%f` or `%g` are used. The latter seems provides the most aesthetically pleasing output. To explicitly express a float constant, append either an f or F to the end, e.g. `12.5f`.
- `double`, used when range provided by float is not sufficient. To display a double value, format character `%f%e%g` can be used.
- `char`: be certain you remember the single quotation and double quotation meant for different type is different in C: single quotation is for char type; double quotation is for character string.
- `_Bool`, this is defined in `stdio.h`. You can also include `stdbool.h`, which defines type `bool` and value `true` and `false`. Therefore, you can declare vairables to be of type `bool` instead of `_Bool`, for only cosmetic purposes.

1.2 `typedef` statement

First, an example:

```
typedef int Counter;
Counter i,j;
```

The main advantage of the use of the `typedef` in this case is in the added readability - it shows intended purpose of the new type. Some other examples:

```
typedef char Linebuf [81];
Linebuf text, inputLine;  # => char text[81], inputLine[81];

typedef char * StringPtr;
StringPtr  buffer;
```

The first example defines a type called `Linebuf`, which is an array of 81 characters; The second example defines a type name `StringPtr` to be a `char` pointer.

If the above definition seems a bit unnatural, follow the steps below:

- write the statement as if a variable of desired type were being declared
- Where the name of the declared variable would normally appear, substitute that new type name
- In front of everything, place the keyword `typedef`

1.3 Variable

A local (defined within a function) auto variable and static variable differ in at least two aspects:

- first, the value of auto variable disappears each time a function finishes execution, while the value of static variable *stays*.
- second, a static local variable is initialized only *once* at the start of overall program execution - and not each time the function is called. Furthermore, the initial value specified for static variable must be a simple constant or constant expression. Static variables also have default initial value zero, unknow automatic variable, which have no default initial value.

1.4 Control Structure

- The if else statement:

```
if (a >= 95)
    printf("Excellent\n");
else if ( a >= 80)
    printf("Good\n");
else if (a >= 60)
    printf("Pass\n");
else
    printf("Fail\n");
```

- Loop statement

```
for (i = 0; i < 100; i++)
    total += i;

for (int counter = 1; counter <=5; ++counter)
   total + = counter;
```

- Loop variant

```
i = 0
while (i < 100)
    total += i;

i = 0
```

```
do {
    total += i;
} while ( i < 100)
```

- Switch statement

```
switch ( expression )
{
        case value1:
                ...
                break;
        case value2:
                ...
                break;
        default:
                ...
                break;
}
```

- Conditional operator

```
condition ? expression 1 : expression 2
```

The condition is an expression, usually a relational expression, that is evaluated first. If the result of evaluation of condition is TRUE (non-zero), then expression-1 is evaluated and the result of the evaluation becomes the result of the operation. If condition evaluates FALSE, then expression-2 is evaluated and its result becomes the result of the operation. Example: `maxValue=(a>b)?a:b`: this statement assigns to the variable `maxValue` the maximum of `a` and `b`.

2 Array

- Declaring an array: (1) declare the type of elements to be contained in the array, (2) the maximum size of the array: `floataverages[200]`.
- Initialize array:

```
int values[10] = {0, 1, 2, 3} /* the rest of elements are 0 */

int M[4][5] = {
        { 10, 5, -3},
        {9, 0, 0 },
        {32, 30, 1},
        {0, 0, 8}
};
```

- Variable length array

  This is allowed, to declare a variable used as the maximum length of the array: for example:

```
int i, numFibs;
printf("How many Fibonacci numbers you want?");
scanf("%i", &numFibs);
unsigned long long int Fibonacci[numFibs];
...
```

3 Function

- Function declaration

```
double power(double val, unsigned pow)
{
    double ret_val = 1.0;
    unsigned i;
    for (i = 0; i < pow; i++)
        ret_val *=val;
    return(ret_val);
}
```

  A function must be **declared** before it can be used or called. This is more common when you call a function defined in a different file for example. You can also make prototype declaration for all functions at the beginning of the file. To play safe, declare all functions in your program, even they are defined before called.

  If a function takes a variable number of arguments, such as `printf`, the compiler must be informed:

```
int printf (char * format, ...);
```

- Function and Array: to pass array element into a function, nothing special: the value of the array element is **copied** into the value of the corresponding formal parameter when the function is called. Passing an entire array to a function: it is only necessary to list the name of the array, without any subscripts. One of key difference is, when you operate on array element inside the function, you will actually modify the value of original passing array!

  The explanation is, when dealing with arrays, the entire contents of the array are *not* copied into the formal parameter array. Instead, the function gets passed information descrbing *where* in the memory the array is located.

  Most of the time, you want the number of array element to be flexible, so you might want to pass in the number as the second argument. Therefore, the following declaration is acceptable:

```
int minimum(int values[], int numOfElements) { ... }
```

All the compiler is concerned with is the fact that an array is expected as an argument to the function and not how many elements are in it.

4 Pointers

Say we have an int called `i`. It's address could then be represented by the symbol `&i`. We can save this address in a pointer (pointing to int) by:

```
int *pi = &i;
```

`int *` is the notation for a pointer to an int. `&` is the operator which return the address of its argument. We say `&i` as referencing `i`, and `*pi` as de-referencing `i` as it gives the value of `i`.

4.1 Structure and Pointers

You can define pointer that points to structures:

```
struct date todaysDate;
struct date *dataPtr;
dataPtr = &todaysDate;
(*dataPtr).day = 21;
```

The last statement has the effect of setting the day of the `date` structure pointed by `datePtr` to 21. The parentheses are required. Pointer to structure are so often used in C that a special operator exists in the language. The structure pointer operator `->`, permits expressions that would otherwise be written as `(*)x.y` to be more precisely expressed as `x->y`.

Structures containing pointers

Naturally, a pointer also can be a member of a structure, for example, a linked list is defined as:

```
struct entry {
    int value;
    struct entry * next;
}

struct entry n1, n2;
n1.next = &n2;
n2.next = (struct entry *) 0;  # mark the end with null pointer
```

4.2 Keyword `const` and Pointers

The following case reads as "charPtr is a constant pointer to a character":

```
char * const charPtr = & c;
charptr = & d;   # invalid
```

The following case reads as "charPtr points to a constant character":

```
const char *charPtr = &c;
*charPtr = 'Y';  # invalid
```

In the case in which both pointer variable and the location it points to will not be changed through the pointer, the following declaration is used:

```
const char * const *charPtr = &c;
```

4.3 Array and Pointers

Using pointer to arrays generaly result in code that use less memory and more efficient. For example, if you have an arrray of 100 integers called `values`, you can define a pointer called `valuePtr`, which can be used to access the integers contained in this array:

```
int *valuePtr;
valuePtr = values      # option 1: set valuePtr point to first elment
valueptr = &values[0]; # option 2: of the array
```

When you define a pointer that is used to point to the elements of an array, you don't designate the pointer as type 'pointer to array', rather, you designate the pointer as pointing to the type of element that is contained in the array. Another example, say you have an array of characters called `text`, you can define a pointer to be used to point to elements in `text` with statement like:

```
char *textPtr;
```

The real power of using pointers to array comes into play when you want to sequence through the elements of an array: `*(valueptr+i)` can be used to access the value contained in `values[i]`. Another common idiom is `valuePtr+=n` set pointer `n` elements farther in the array - so general that no matter what type of element is contained in the array.

```
/* calculate the sum of elements contained in array */
int arraySum (int array[] , const int n) {
    int sum = 0, *ptr;
    int * const arrayEnd = array + n;
    for (ptr = array; ptr < arrayEnd; ++ptr)
        sum += *ptr;
    return sum;
}
```

An array is actually a pointer to the 0th element of the array. De-referencing the array name will give the 0th element.

Array Access	Poiner Equivalent
`arr[0]`	`*arr`
`arr[2]`	`*(arr + 2)`
`arr[n]`	`*(arr + n)`

Since an arrary is like a pointer, we can pass an array to a function, and modify its element without concerning reference and de-reference. A function which expects to be passed an array can declare that parameter in two ways:

```
int arr[];          char **argv; /* array of strings*/
or                  or
int **arr;          char *argv[];
```

Realizing now you could have declared `array` to be `int` pointer in the preceding program example, and then could have subsequently used it as such, we can eliminate the `ptr` variable:

```
int arraySum(int *array, const int n) {
    int sum = 0;
    int * const arrayEnd = array + n;
    for ( ; array < arrayEnd; ++array)
        sum += *array;
    return sum;
}
```

4.4 Character String and Pointers

Whenever a constant character string is used in C, it is a pointer to that character string that is produced. So, if `textPtr` is declared to be a character pointer, as in

```
char *textptr;
```

then the statement

```
textPtr = "A character string";
```

assigns to `textPtr` a *pointer* to that constant string. Be careful to make the distinction between character pointer and character arrays, as the type of assignment just shown is not valid with character array:

```
char text[80];
text = "A character string"; # not valid
char text[80] = "This is OK";
```

The only case that C lets you get away with performing this type of assignment is shown at the last statement above.

Here is another example to illustrate pointer to character strings:

```
void copyString(char *to, char *from) {
    for ( ; *from != '\0'; ++from, ++to )
        * to = *from;
    *to = '\0';
}
```

One example to illustrate operations you can perform on a pointer: you can subtract two pointers of the same type, and return number of elements contained between two pointers:

```
/* calculate length of a string */
int stringLength(const char *string) {
    const char *cptr = string;
    while (*cptr)
        ++cptr;
    return cptr-string;
}
```

4.5 Function and Pointers

To declare a pointer to a function, do:

```
int (*pf) ();
```

This simply declare a pointer `*pf` to function that returns an int. No actual function is pointed to, yet. If we have a function `int f()` then we may simply write:

```
pf = &f;
```

You can then do things like:

```
ans = f(5);
ans = pf(5);
```

which are equivalent.

Here is how it is used in practice. Quick sort `qsort` is prototyped in `stdlib.h` as:

```
void qsort(void *base, size_t num_elements, size_t element_size,
    int (*compare)(void const *, void const *));
```

The argument`base` points to the array to be sorted, `num_elements` indicates the length of the array, and `element_size` is the size in byte of array element. Final argument `compare` is a pointer to a function.

qsort calls the compare function which is user defined to compare the data when sorting. Note that qsort maintains it's data type independence by giving the comparison responsibility to the user. The compare function must return certain (integer) values according to the comparison result:

```
< 0: if first value if less than the second
= 0: if they are equal
> 0: if first value is greater than the second
```

Some quite complicated data structures can be sorted in this manner. For example, to sort the following structure by integer key:

```
typedef struct {
        int   key;
        struct other_data;
} Record;
```

We can write a compare function, `record_compare`:

```
int record_compare(void const *a, void  const *b)
  {  return ( ((Record *)a)->key - ((Record *)b)->key );
  }
```

Assuming that we have an array of `array_length` Records suitably filled with date we can call qsort like this:

```
qsort( array, arraylength, sizeof(Record), record_compare);
```

4.6 Event Handler

In C, each function has an address in the code segment. This address maybe stored in a pointer (with special syntax), and later invoked from this pointer. This is the mechanism that is commonly used in callbacks and virtual function tables.

```
void (*event_handler)(Event*) = NULL;

void my_event_handler(Event *event) {
     printf("An event of type:%s occurred\n", event->type);
}

void do_tasks() {
   Event event;
   /* ....... Do some tasks */
   /* if event occurs */
   if (event_handler != NULL) {
     event_handler(event);
   }
}

main() {
    event_handler = my_event_handler;
    do_tasks();
}
```

5 Structure

To define a structure:

```
struct date
    int month;
    int day;
    int year;
}
```

This definition of `date` in a sense defines a new type in the language in that variables can subsequently be declared to be of type `structdate`, as in the following:

```
struct date purchaseDate
```

You can initialize a structure in several ways:

```
struct date today = { 7, 2, 2005};
struct date today = { .month=9, .day=25, .year=2004};
today = (struct date) { 9, 25, 2004 };
today = (struct date) {.month=9, .day=25, .year=2004};
```

By giving explict member name, you can specify it in any order, even partially. You can define and declare and intialize at the same time as well:

```
struct date {                   or       struct date {
  int month;                                 ...
  int day;                                   ...
  int year;                                  ...
} todayDate = { 1, 11, 2005};             } dates[1000];
```

Array of Structures

```
struct date birthday[15];
struct date myfamily[3] = { {2,3,2004},{3,4,2005}};
```

6 String Processing

In C, the special character that is used to signal the end of a string is known as *null* character and is written as `'\0'`. Example: count the characters in a string:

```
int stringLength(const char string[])
{
    int count = 0
    while (string[count] != '\0')
        ++count;
    return count;
}
```

As far as character array initialization goes, you can certainly do it this way:

```
const char word[] = {'a','s','t','e','r','\0'};
```

or you can initialize it by simply specifying a constant character string rather than a list if individual characters:

```
char word[] = { ''hello''};   /* auto terminated by null character */
```

Also in C, you can't tell string equality by using == operator such as:

```
if (string1 == string2)
...
```

as == are only defined for simple variable types.

A final example: convert string to integer, for example, '245' will return integer value 245.

```
int strToInt ( const char string[])
{
    int i, intValue, result = 0;
    for (i = 0; string[i] >= '0' && string[i] <= '9'; ++i)
    {
        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }
    return result;
}
```

7 Working with Multiple Files

In the situation where A function in a file calls B function in another file, you should always *make certain to include a prototype declaration so taht compiler knows the function's argument types and the types of the return value*.

7.1 External variable

Function contained in separate files can communicate through *external* variables: they can be accessed and changed by multiple modules (files). You need to follow a few rules when working with external variable:

- First, you must define it someplace in your source file. You can define it in two mutually exclusive ways, one is:

  ```
  int shard_var;
  ```

  This statement should be outside any function. Here, an initial value can be optionally assigned to the variable. The second way of defining external variable it to declare the variable outside any function, as in first case, but placing keyword `extern` in front of declaration, PLUS, **explicitly assigning an initial value to it**, as follows:

  ```
  extern int shared_var = 0;
  ```

  Again, we note that these two approach are mutually exclusive.

- The second rule to follow is, after you define an external variable somewhere as above, you must declare your intention of using it by putting the following statement in the module where you intend to access it, as follows:

  ```
  extern int shared_var;
  ```

  You can put above statement inside a function if the access is local to that function only. Or you can put it outside any function in the case where multiple functions are accessing it.

7.2 Global vs. External

Above discussion suggests taht any variable defined outside a function is not only a global variable, but is also an external variable. In many cases, you would want to define a variable to be global but *not* external. In other word, you want to define a global variable to be local to a particular module (file) - make sense in cases where no functions other than those contained inside a particular file need access to that particular variable. This can be done by defining a variable to be `static`.

```
static int shared_var = 0;
```

The static declaration more accurately reflects the variable's usage and avoid conficts created by two modules using the same name unknowingly for global variable.

7.3 Static functions

Unlike variables, no special mechanisms are required to call function defined in another file - that is, you don't need `extern` declaration for that function.

When a functino is defined, it can declared to be `extern` or `static`, the former case being the default. A static function can be called only from within the same file as the function appears. For example:

```
static double squareRoot (double x)
{ ... }
```

8 Misc Features

8.1 Common Operator

The comma operator can be used to separate multiple expressions anywhere that a valid C expression can be used. The expressions are evaluated from left to right. So in the statement:

```
while (i < 100)
   sum += data[i], ++i
```

the value of `data[i]` is added into `sum` and then `i` is incremented. Because all operators in C produce a value, the value of the comma operator is that of the rightmost expression.

8.2 Type Qualifier

The following qualifiers can be used in front of variables to give compiler more information about the intended use and in some cases, generate better code.

- `register` qualifer: request a particular variable to be accessed as fast as possible. One difference to note is that you can not apply address operator to such variable, other than that, there is no difference from the automatic variable.

  ```
  register int index;
  register char *textptr;
  ```

- `volatile` qualifer: It tells the compiler explicitly that the specified variable *will* change its value, and don't do optimization. Suppose in some I/O program, you want to write two character to a port:

  ```
  *outPort = 'O';
  *outPort = 'N';
  ```

  Some smart compiler will notice that two assignment to same location, and because `outPort` is not modified in between, it may simply remove the first assignment from the program. To prevent this from happening, we declare: `volatilechar*outPort`.

8.3 Dynamic Memory Allocation

The standard C libary provides `malloc` and `calloc` for allocating memory at runtime. The `calloc` takes two arguments that specify the number of elements to be reserved and size of each element in *byte*. The function returns a pointer to the beginning of the allocated storage area in memory. The storage area is also automatically set to 0. `malloc` works the similarly, except it only takes a single argument - the total number of bytes of storage to allocate, and it doesn't set to zero automatically.

Remember both `malloc` and `calloc` are defined to return a pointer to `void` and, as noted, this pointer should be type cast to the appropriate pointer type. For example:

```
int * intPtr;
intPtr = (int *) calloc( sizeof(int), 1000);
if (intPtr == NULL) {
    fprintf(stderr, "calloc failed\n");
    exit (EXIT_FAILURE);
}
```

Another example: adds a new entry to a linked list:

```
struct entry *addEntry (struct entry *listPtr) {
    // find the end of the list
    while (listPtr->next != NULL)
              listPtr = listPtr->next;
```

```
    // get storage for new entry
    listPtr->next = (struct entry *) malloc(sizeof(struct entry));
    // add null to the new end of list
    if (listPtr->next != NULL)
       (listPtr->next)->next = (struct entry *) NULL;
    return listPtr->next;
}
```